Association for Information Systems
AIS Electronic Library (AISeL)

ECIS 2012 Proceedings

European Conference on Information Systems (ECIS)

5-2-2012

A CONSOLIDATED UNDERSTANDING OF TECHNICAL DEBT

Edith Tom
University of New South Wales

Aybuke Aurum
University of New South Wales

Richard Vidgen
Hull University

A CONSOLIDATED UNDERSTANDING OF TECHNICAL DEBT

Tom, Edith, School of Information Systems, Technology and Management, University of New South Wales, Sydney, NSW, 2052, Australia, z3251794@student.unsw.edu.au

Aurum, Aybüke, School of Information Systems, Technology and Management, University of New South Wales, Sydney, NSW, 2052, Australia, aybuke@unsw.edu.au

Vidgen, Richard, Department of Management Systems, Hull University Business School, Hull, HU6 7RX, United Kingdom, r.vidgen@hull.ac.uk

Abstract

Technical debt utilises financial debt as a metaphor to describe the phenomenon of increasing software development costs over time. Whilst this phenomenon is evidently detrimental to the long-term success of software development, it appears to be poorly understood in the academic literature. The absence of a clear definition and model for technical debt means that the notion of technical debt remains metaphorical, thus preventing the realisation of technical debt's utility as a conceptual and technical communication device.

This exploratory study reconciles the high-level, abstracted view of technical debt presented in academic literature. It establishes the boundaries of the technical debt phenomenon and develops a comprehensive theoretical framework to facilitate future research. The resulting theoretical framework portrays a holistic view of technical debt that incorporates a set of precedents and outcomes, as well as the phenomenon itself.

Keywords: Technical debt, code debt, healthy-unhealthy debt, systematic literature review,

1 Introduction

Traditionally, project management relies on the management accounting idea of 'variance' analysis to monitor the gap between actual and planned performance with regard to project cost, time, and quality. In this paper we explore 'debt' – specifically, *technical* debt - as an alternative metaphor for the management of software projects. Technical debt is recognised as a critical issue in the software development industry, with the global technical debt bill estimated by Gartner to be $US500 billion with the potential to double in five years' time (Thibodeau, 2011). However, this does not mean that debt is necessarily 'bad'. For example, a small level of debt can help developers speed up the development process in the short term. Regardless, the consequence of this may be felt in the longer term if the project is highly 'geared' (which implies onerous debt repayments), leading to slower development and killing of productivity.

Academic literature reveals a poor understanding of the phenomenon – whilst it suggests that technical debt is known to be detrimental to the long term success of software development projects (Lindgren et al., 2008b, O'Connor, 2010, Neill and Laplante, 2006), there is an absence of any comprehensive definition or conceptual model. This may be due to the fact that the technical debt phenomenon is collectively understood by the software development community through the application of a metaphor. It follows that the boundaries of any definition would be determined by how far we are collectively willing to extend this metaphor. The challenge of defining technical debt in academic literature lies in the fact that such boundaries have not yet been identified through a rigorous process.

There is a need to rigorously define and validate the technical debt concept in academic literature so that its impacts – including any financial implications – can be understood, within the practitioner community as well as in academia. Understanding the constituent parts of technical debt, motivations for allowing technical debt to accrue, and motivations for paying it down is essential to realising the utility of technical debt as a concept and as a communication device.

The objective of this paper is to provide a consolidated understanding of the technical debt phenomenon, to reflect this consolidated understanding in the form of a theoretical framework and discuss the positive and negative outcomes of technical debt. In doing this we aim to move from technical debt as metaphor to technical debt as something more tangible and literal that can ultimately be made operational and used by both project and product managers to improve any type of software project outcomes, for example, by taking on the right amount of technical debt ("gearing" – the ratio of debt to capital) of the right type at the right time. A systematic literature review (SLR) on technical debt has been developed to understand the state of the research addressing this area. This SLR was then used to develop a theoretical framework that reflects a comprehensive and reconciled view of the technical debt phenomenon.

The remainder of this paper is structured as follows. A brief background to the study is presented in Section 2. The SLR process as a research method is detailed in Section 3. The results of the SLR and discussion are presented in Section 4. A theoretical framework and limitations of this study are provided in Section 5, with conclusions drawn in Section 6.

2 Background

Technical debt is a metaphor that highlights the consequences of sloppy software development. The concept of technical debt was first introduced by Cunningham (1993), who described how *"shipping first time code is* **like** *going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite."* (our emphasis). Since then, the suitability of debt as a way of explaining the various drivers of increasing costs throughout the life of a software system has been affirmed by the software development

community, with the debt metaphor being used to make observations regarding these increasing development costs in literature. The concept of technical debt provides a framework for thinking about how to manage the effort that goes into the software development process (Shull, 2011). Hence, managers should understand this concept, measure it and communicate it to the stakeholders such that software developers and stakeholders have shared understanding of what constitutes technical debt and how/when it should be entered in to and when it should be prevented.

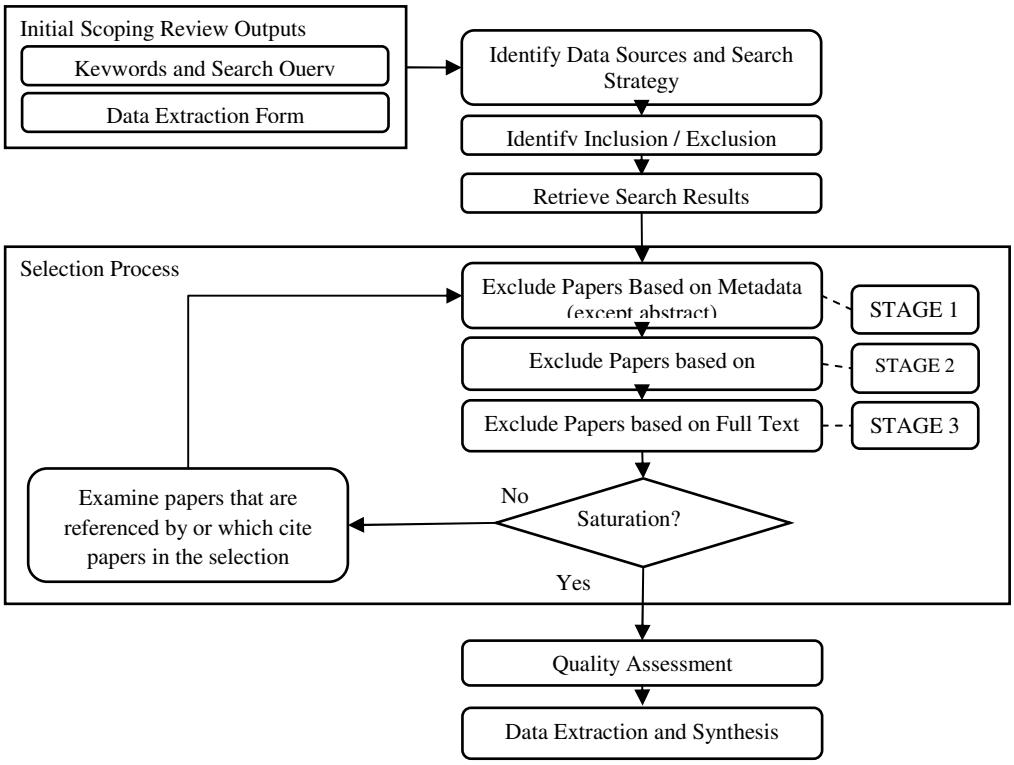

Figure 1: Systematic Literature Review Process

3 Research Method

SLR is a methodical way to identify, evaluate, and interpret the available studies conducted on a topic, research question, or a phenomenon of interest (Kitchenham, 2004). As an initial, ad-hoc literature review suggested that extant academic literature on technical debt is sparse, SLR based on Kitchenham's (2004) guidelines, was conducted to more rigorously ascertain the state of the art and to identify the gaps in current research in a thorough manner. The following questions were developed to guide the SLR.

- RQ1 – What are the elements of technical debt?
- RQ2 – Why does technical debt arise?
- RQ3 – What are the benefits and drawbacks of allowing technical debt to accrue?

The following procedure was followed in conducting the SLR:

1. Bibliographic databases e.g. Scopus, Inspec, Web of Science, were selected from which to search for publications. Further information can be found in Tom, 2011. The holdings of these databases were checked to ensure that top information systems journals as identified by the AIS Senior Scholars' Bucket (n.d.) and software engineering journals (Wong et al., 2011) were covered by the search.

2. Keywords describing the research area were identified, and search strings were created for the selected databases. An initial scoping review identified the terms "design debt" and "debt metaphor" to synonymously reference the phenomenon of technical debt. Using these keywords the search strings were run against to each databases.
3. The list of publications from each database was combined with duplicate records for a publication removed. The title and abstract of each publication was then independently reviewed by the three authors against a set of inclusion and exclusion criteria. Search results were excluded if they a) did not relate to the process of software development; b) did not discuss the concept of technical debt; c) could not be accessed through the holdings of the University of New South Wales library.
4. Papers selected for inclusion in the review as a result of this process are listed (see References with *). Results were then cross-checked for potential bias, and the union of these results progressed to the next stage. Backward and forward searches were also conducted; such that papers referenced by or which cited each paper were included as potential candidates to feed back into the systematic literature review process. Two major citation databases, Scopus and Web of Science were used to source data for the forward search.
5. An initial scoping review revealed an absence of empirical studies examining technical debt, negating the benefit of assessing quality in terms of study design. This was confirmed by the data extraction process. A decision was made to instead classify papers as either academic or non-academic, excluding the latter from SLR at this stage. A paper was considered academic if its source was listed in the ERA 2010 (Excellence in Research for Australia) journal and conference rankings released by the Australian Research Council (2010), or if the source employed a formal peer-review process for assessing content for publication.
6. Finally a data extraction was undertaken on the relevant publications. A form was designed to extract the necessary data to address the SLR questions (Tom, 2011). A distinction was made between implied and explicitly stated meanings. Any contradictions to purported elements, causes and consequences of technical debt were also noted. The appendix shows the themes covered in this SLR.

A summary of this process is provided in Figure 1.

	Initial	**Stage 1**	**Stage 2**	**Stage 3**
First iteration	194	98	87	**18**
Backward search	254	237	10	**1**
Forward search	23	17	2	**0**
			Totals	**19**

Table 1 - Papers at Each Stage of the Systematic Literature Review Selection Process

4 Results and Discussion

A total 19 publications were identified and analysed as part of the SLR. As shown in Table 1, the review of extant academic literature reveals a limited number of articles that refer to the concept of technical debt. Results of the data extraction process revealed an apparently fragmented understanding from a number of disparate themes and anecdotal or otherwise high-level definitions that fail to describe the phenomenon comprehensively or in detail. Examples of such definitions include *"the degree of incompleteness"* (Klein, 2005), *"a backlog of deferred technical problems"* (Torkar et al., 2011), and *"any side of the current system that is considered sub-optimal from a technical perspective"* (Ktata and Lévesque, 2010). Whilst more specific definitions such as *"bugs, design issues, and other code-quality problems"* (Black et al., 2009) can be found in the literature, a comparison of the specific elements of technical debt purported in each definition reveals disparity. This is affirmed by Brown et al. (2010a) who recognise that, beyond

what is intuitively understood, *"a more rigorous definition and validation of the* [technical debt] *concept, and the heuristic practices it implies has not been undertaken"*.

4.1 Elements of Technical Debt

The findings showed that there is wide agreement that code decay and design debt form two elements of technical debt. However, it is also apparent that the boundaries of technical debt remain unclear. It can be seen that there is not yet a comprehensive, granular definition of the phenomenon in terms of its elements, presenting a barrier to its effective identification and management.

4.1.1 Code decay and design debt

Academic literature reflects much agreement that the concept of technical debt encompasses code-, design- and architecture-related aspects of a system. These can be considered to be the more certain literal elements of the technical debt phenomenon.

Code decay as a form of debt is implicit in Cunningham's (1993) coining of the debt metaphor – *"Every minute spent on not-quite-right code counts as interest on that debt"*. Conceptually, the properties of "decayed" code are very similar to those of an entire system suffering technical debt – *"more difficult to change than it should be"* in terms of personnel cost, time, and the level of achievable quality (Eick et al., 2001). Unlike the overall phenomenon of technical debt, its code decay subset is well defined. Eick et al., (2001) present a conceptual model describing the symptoms of decayed code, which are echoed by many papers that discuss the code aspect of technical debt. Examples include unnecessary coupling (Smith, 2009), a lack of abstraction (Klein, 2005), code duplication (Shull, 2011, O'Connor, 2010), poor evolvability (Mantyla and Lassenious, 2009), and quick rather than elegant solutions (Torkar et al., 2011). Code decay is also directly acknowledged to be a type of technical debt (Brown et al., 2010a, Shull, 2011, Neill and Laplante, 2006).

The concept of design debt refers to the effect when software development occurs without giving sufficiently rigorous consideration to the system's original design and architecture, and without structural refactoring as necessary (Neill and Laplante, 2006). In contrast to code decay, the definition of design debt remains at a high-level. However, it is also widely supported as an aspect of technical debt. Degraded software design that deviates significantly from reference architecture (Shull, 2011), design based on *"less-than-optimal"* choices (Wirfs-Brock, 2008b), and excessive design complexity (Lutz, 1993) are all listed as elements of technical debt.

4.1.2 Fuzzy boundaries of technical debt

Compared to code decay and design debt, literature presents less certainty about a range of elements from the existence of known defects through to a lack of documentation. This is hardly surprising, considering that the phenomenon technical debt describes is identified through the application of a metaphor – *"shipping first time code is* like *going into debt"* (Cunningham, 1993). Whilst the academic community is willing to extend the metaphor of financial debt to technical issues of code decay and design debt, the precise boundaries of our understanding of the technical debt phenomenon are yet to be defined. Brown et al. (2010a) propose that technical debt can be characterised as the *"gap between the current state of a software system and some hypothesized 'ideal' state in which the system is optimally successful in a particular environment"*, suggesting that known defects, unimplemented features and outdated documentation can all be considered aspects of debt. Whilst visualising technical debt as a gap is a useful conceptual tool, it is essentially another metaphor, and does not further clarify what actually constitutes technical debt in a literal sense. However, the additional purported elements of debt provide insights into what the collective understanding of technical debt might include.

Klein (2005) acknowledges that defects, issues and unimplemented features are a form of debt by identifying that technical debt may be considered a representation of a *"degree of incompleteness"* for change introduced to a system. Such incompleteness would include unhandled scenarios such as edge cases, and known performance issues. Black et al. (2009) echo the notion that bugs are a form of technical debt. There is also some agreement that outdated or non-existent documentation contributes to increasingly costly change to a system (Shull, 2011, O'Connor, 2010).Yet another purported component of technical debt is a lack of, or incomplete state of system testing. Deferred testing increases the risk of system failures, security issues and other defects - additional verification and validation is touted to be a common corrective action for technical debt (Shull, 2011). O'Connor (2010) also refers to the notion of testing debt, stating that unit tests are often sacrificed as pressure to meet deadlines mount. The metaphorical interest on such testing debt is perceived to be a shrinking safety net for catching regression bugs (O'Connor, 2010).

While all of the aforementioned aspects of a system, and more – Debois (2008) suggests that the postponement of infrastructure updates also leads to the accrual of technical debt – seem to fit the metaphor, they are not widely acknowledged in academic literature. It is evident that the boundaries of technical debt as reflected in academic literature are *fuzzy* – lacking clarity and definition. These fuzzy boundaries are a barrier to efforts to model, quantify and manage technical debt (Brown et al., 2010a).

4.2 Why does Technical Debt Arise?

The fuzzy boundaries of technical debt pose a challenge to its monitoring and management. This challenge is amplified by the motivations behind allowing technical debt to accrue. The low visibility of both the elements of technical debt and the phenomenon as a whole due to its fuzzy boundaries encourages decision-making that does not consider the adequate management of debt, but is instead driven by project constraints.Smith (2009) argues that the traditional three-legged stool analogy of project management, which advocates scope, resources and time as the binding constraints of a project, should be supplemented by a "fourth leg" – quality, and that *"when quality is simply assumed, then bad things can happen and they usually show up in the form of technical debt"*. Indeed, a review of academic literature reveals that budgeting and resourcing constraints are thought to be contributors of technical debt, and that the constraint of time in particular is a significant recurring theme. A lack of budget and development resources (O'Connor, 2010) for projects and the constant pressure to *"do more with less, hit timelines, show return on investment, and meet commitments"* (Smith, 2009) are claimed to invariably result in the accumulation of debt. There is wide agreement that the pressure of deadlines can lead to shortcuts in code and architectural maintenance, and that prioritisation of urgent tasks in the short term displaces those that are important in the long term, such as the repayment of technical debt (Torkar et al., 2011, Lindgren et al., 2008b, Brown et al., 2010a). Even when deadlines are comfortable, projects can acquire a sense of *"completion frenzy"*, whereby the principle of allowing business value to drive development is adopted in a simplistic manner (Heidenberg and Porres, 2010). It is often the case that technical tasks are prioritised by the business team (Davis and Andersen, 2009), and product management don't necessarily have the skills to consider architectural issues when planning for releases (Lindgren et al., 2008b). A lack of pre-emptive allocation of dedicated effort to reduce technical debt ensues (Torkar et al., 2011), increasing the difficulty of paying back the debt as it accumulates.

Prioritisation of effort to reduce technical debt is further made difficult by its poor visibility. This is in part due to an incomplete understanding of its intrinsic nature– the fuzzy boundaries of what constitutes technical debt. Development teams find it difficult to justify the cost of technical debt in terms of business value (Davis and Andersen, 2009), but often need to when it means that features need to be de-scoped (Heidenberg and Porres, 2010). There is an *"urgent need to quantify the* [technical] *debt accumulated over time"*, so that it can be communicated to product owners (Ktata and Levesque, 2010).

4.3 Benefits and Drawbacks of Allowing Technical Debt to Accrue

The motivations for allowing technical debt to accrue amplify the challenge that a limited understanding of the phenomenon creates in managing the repayment of debt. These motivations are driven by the pressure of project constraints, and limited resources(Shull, 2011). This highlights the need to fully understand the benefits and drawbacks of allowing technical debt to accrue, so that the decision to ignore the accumulation of debt becomes conscious and informed. Table 2 summarises the benefits and drawbacks of technical debt mentioned in academic literature. Overall, the significance of the consequences discussed in this section emphasise an importance and urgency to gain a complete understanding of the technical debt phenomenon.

Benefits:	Drawbacks:
• Decreased costs and faster development in the short term, resulting in less time-to-market (Shull, 2011, Smith, 2009, Brown et al., 2010a, Lindgren et al., 2008b, Lutz, 1993)	• Increasing costs over time, such as the amount of effort required to deliver a certain amount of functionality (Klein, 2005, Lindgren et al., 2008a, Lutz, 1993, Heidenberg and Porres, 2010, Smith, 2009, Shull, 2011) • Work estimation becomes difficult (Davis & Andersen 2009, O'Connor 2010) • Developer productivity is negatively impacted (Black et al., 2009, O'Connor, 2010, Torkar et al., 2011) • Becomes increasingly difficult to repay as decisions are affected by existing debt (Wirfs-Brock, 2008b) • Increased risk involved in modifications to the system (O'Connor, 2010) • Change becomes prohibitively expensive to the point of bankruptcy, and a complete rewrite and new platform may become necessary (Black et al., 2009, O'Connor, 2010) • Decreased quality in the end product (Heidenberg and Porres, 2010, Neill and Laplante, 2006, Lindgren et al., 2008b)

Table 2 - Benefits and Drawbacks of Allowing Technical Debt to Accrue

4.3.1 Healthy vs. unhealthy debt

Ktata and Levesque (2010) describe technical debt that is the result of good intentions, such as *"doing the simplest things that could work and resisting the temptation to predict the future"* as a form of healthy debt, which – whilst still incurring an interest that needs to be repaid in the future – should be considered a business opportunity (See Table 3). On the other hand, technical debt that implements requirements *"to the prejudice of quality"* is considered unhealthy, and triggers both short (defects) and long term (rigidity) costs (Ktata and Levesque, 2010). Shull (2011), who shares this view acknowledges that the phenomenon of technical debt can often be a deliberate investment to speed development in the short term. Another classification comes from Fowler (2009) who advocates that technical debt can be *prudent* or *reckless*, and *deliberate* or *inadvertent*. This type of categorisation provides insights into how the causes of technical debt can be managed. For example, technical debt that is inadvertent and prudent is inevitable, whilst efforts should be made to prevent the accrual of reckless debt.

4.3.2 Interest and bankruptcy

The phenomenon of technical debt conforms to the financial debt metaphor by exhibiting the behaviour of incurred interest – *"in the form of increased future costs"* (Brown et al., 2010a), and potential bankruptcy when a complete rewrite and new platform are necessary (O'Connor, 2010). Whilst the concept of bankruptcy from technical debt is uncommon in academic literature, it is clear that as technical debt accumulates, the cost of changing a system may eventually outweigh the cost of a complete rewrite.

There is much agreement that additional costs comparable to interest on financial debt are a part of the technical debt phenomenon. Klein (2005) notes that a drawback of allowing technical debt to accrue is the increasing amount of effort required to deliver a certain amount of functionality. Indeed, it is widely acknowledged that the accrual of technical debt invariably creates extra work and greater costs in the future compared to dealing with issues on a timely basis (Shull, 2011, Smith, 2009, Lutz, 1993, Lindgren et al., 2008a). Wirfs-Brock (2008b) explains that technical debt becomes increasingly difficult to repay as decisions are affected by existing debt, which may include *"less-than-optimal"* design. Lindgren et al. (2008b) further note that it may be more economic for a company to focus on consistently paying back debt, rather than *"toggling behaviour with full focus on features or quality"* in their attempt to manage the technical debt phenomenon.

Healthy Debt:	**Unhealthy Debt:**	**Source:**
"Technical debt gives us a framework for thinking about the fact that not doing some good things today, no matter how valuable they seem on their own merits, allows us to invest in other good things"	"If we can't identify a plausible alternative benefit that makes 'going into debt' a viable trade-off, and/or if we can't identify a plausible strategy for paying off that debt"	A1 (Shull, 2011)
"[Healthy debt] is a direct result of doing the simplest thing that could work and resisting the temptation to predict the future… this healthy debt is the kind of work that the [product owner] agreed to delay in order to get value sooner" "As a price to this debt, the [product owners] agree to pay interest when comes the time to liquidate the debt, usually, before shipping a suitable version" "Very often, the trigger of such behaviour is called a business opportunity"	"[Unhealthy debt] is usually a trick used by developers to solve a problem to the prejudice of quality" "In the short term, defects delay the release of the software and in the long term software is difficult to maintain and rigid in the face of changing business need"	A5 (Ktata and Lévesque, 2010)
"A little debt speeds development so long as it is paid back promptly with a rewrite"	"The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt"	A19 (Cunningham, 1993)

Table 3 - Healthy and Unhealthy Debt

4.3.3 Literal consequences of accrued technical debt

There is wide agreement in academic literature that the cost of repayments on both interest and principle for technical debt takes the literal form of detrimental effects on difficulty, risk, and quality. Difficulty and risk surrounding scheduling and timelines increase as projects age and accumulate technical debt due to the negative impact on developer productivity when bug fixes, restructurings, and additional time to understand code are necessary before new changes can be implemented (Black et al., 2009, O'Connor 2010, Wirfs-Brock 2008a). As debt ages, maintenance costs increase (Heidenberg and Porres, 2010, Brown et al., 2010a) to the point where they may become *"prohibitively expensive"* (Black et al., 2009).

Ultimately, systems owing a large amount of technical debt risk becoming *"rigid (hard to change), fragile (each change breaks something else), viscous (doing things right is harder) and opaque (hard to understand)"* (Brown et al., 2010a). There is also the risk that technical debt can lead to decreased utility (Neill and Laplante, 2006) and decreased product competitiveness when both R&D efficiency and time-to-market are negatively impacted by the debt (O'Connor, 2010). Whilst the relationship between technical debt and quality is less commonly explored and elaborated on in academic literature, there is agreement that accrued technical debt decreases the quality of software (Heidenberg and Porres, 2010, Lindgren et

al., 2008b), such that it becomes *"less understandable, maintainable, and more complex"* (Neill and Laplante, 2006), and that technical debt is in fact incurred when quality is sacrificed to meet demands in other aspects of software development (Smith, 2009). Heidenberg and Porres (2010) argue that technical debt is in fact simply the inverse of the design quality of the system, as measured by identifying the presence of programming constructs that indicate poor design. Whilst this view is not reflected by other papers in the systematic review, and does not appear to be a complete portrayal of technical debt (for example, it does not allow for the idea of healthy debt), it confirms that quality is a useful construct when conceptualising the phenomenon.

5 Theoretical Framework and Implications

There is a clear need for research that consolidates academic literature with the collective understanding of technical debt that exists in the software development practitioner community, and to build a conceptual model that can be used to undertake empirical investigation. Academic literature suggests that technical debt can be holistically represented by the precedents and behaviours that cause the phenomenon, metaphorical and literal elements that constitute the phenomenon's intrinsic nature, and a number of outcomes that result from healthy and unhealthy forms of technical debt. As depicted in Figure 2, numerous constructs form the representation of technical debt in academic literature. Whilst some of these constructs are better supported in literature than others, Figure 2 provides an initial theoretical framework.

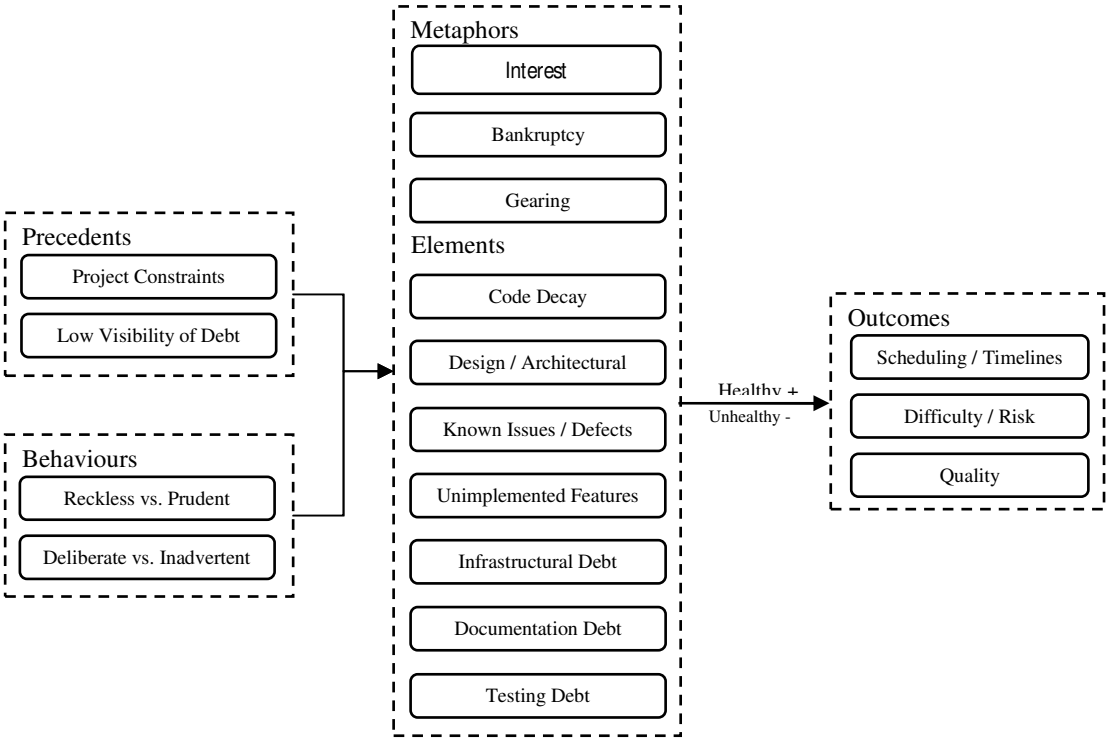

Figure 2 - An Initial Theoretical Framework of Technical Debt

As a conceptual device, the framework can be leveraged to inform action in response to perceived technical debt, and as a comprehensive guide when assessing software development practices and the status quo. As a communication device, the framework can be used to aid developers in flagging issues that are rarely visible or considered at an executive level. For the academic community, this research contributes fundamental research that addresses a current gap in understanding. Current thinking about

technical debt is abstract, high-level and anecdotal. There is an absence of a comprehensive definition, conceptual model, or complete theoretical framework of technical debt in academic literature. The theoretical framework presented in this study explicitly identifies the literal elements of software development technical debt is manifest, which can inform and enable empirical studies investigating technical debt to be conducted. For the practitioner community, this theoretical framework helps to realise the utility of technical debt as a tool for conceptualisation, communication, and management. The framework presents the technical debt concept in a more rigorously defined and comprehensive form. The framework facilitates more effective identification and acknowledgement of technical debt by highlighting aspects of software development where the potential for technical debt's existence might have been overlooked by individuals and teams. Additionally, it can assist in making technical debt visible and accounted for, by allowing developers to more effectively communicate technical problems to management, and for managers to make better-informed decisions concerning technical debt.

6 Conclusion

This study focused on establishing a fundamental theoretical framework of technical debt, based on extant academic literatures and provided answers to three research questions. The SLR results reflected several recurring themes that constitute an understanding of technical debt, but there is a complete absence of primary studies examining the phenomenon as a construct. Furthermore, many of the papers identified to be sufficiently relevant for inclusion in this review present an abstracted view of technical debt through implicit meanings and a reluctance to define its properties and boundaries.

The findings showed that whilst code decay and architectural deterioration are commonly recognised to be major elements of technical debt, many other concepts including a lack of documentation and testing are yet to be widely recognised and agreed upon as forms of debt. It becomes apparent from the literature that recognising, quantifying and accurately communicating the extent of technical debt in a system poses a significant challenge. Regardless, the motivations for the accrual of technical debt are clear and compelling. Time as a constraint is a recurring theme in the technical debt literature, being impacted by, and at the same time a factor of, technical debt. Other project constraints such as budgeting and resourcing are also drivers for allowing technical debt to accrue, assisted by the low visibility of the phenomenon. The notion of debt as a type of investment that enables an increase development velocity in the short term creates the temptation to allow it to accrue, despite it inevitably slowing velocity and increasing time-to-market in the long term if it is not repaid. Thus, it becomes important to gain a complete understanding of technical debt and the actual trade-offs (obvious and otherwise) that are made when it is allowed to accrue. This importance is emphasised by risks comparable to interest and potential bankruptcy, along with the literal consequences (good and bad) underpinning the debt metaphor. These benefits and drawbacks of allowing technical debt to accrue reveal points of agreement and disagreement, and ultimately confirming a need for further research into questions such as: what types of debt are healthy? When should debt be entered into? What levels of debt are sustainable (gearing)? Whilst a protocol was developed to support this SLR, some limitations exist surrounding protocol decisions. The data sources and search strategy of this review aimed to ensure that all relevant articles were included by using keywords identified in an initial scoping review, it cannot be claimed that this – or any – SLR is entirely complete. Additionally, there is the risk that the phenomenon described by technical debt might be described with different terminology that is not covered by the search terms of this review. Whilst this risk was addressed by incorporating a scoping review and forward search into this review, it remains as an acknowledged validity threat.

The researchers are currently evaluating this framework through interviews with academics and software practitioners. Ultimately, future research should incorporate empirical studies to validate heuristics and techniques that will assist practitioners in their management of technical debt.

References

Senior Scholars' Basket of Journals [Online]. Association for Information Systems. http://home.aisnet.org /displaycommon.cfm?an=1&subarticlenbr=346 [Accessed March 2011].

ERA 2010 [Online]. Australian Research Council. http://www.arc.gov.au/era/era_journal_list.htm.

Black, S., Boca, P.P., Bowen, J.P., Gorman, J. and Hinchey, M. (2009). Formal Versus Agile: Survival of the Fittest. Computer, 42**:**37-45. -***A8**

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K. and Zazworka, N. (2010a). Managing Technical Debt in Software-Reliant Systems. FSE/SDP Workshop on the Future of SE Research, November 7-11, 2010, Santa Fe, NM, USA. ACM, 47-51. - ***A3**

Brown, N., Nord, R. and Ozkaya, I. (2010b). Enabling Agility through Architecture. CrossTalk**,** 12-17.

Cunningham, W. 1993. The WyCash Portfolio Management System. OOPS Messenger, 4(2): 29-30. ***A19**

Davis, J.D. and Andersen, T.J. (2009). Surviving the Economic Downturn. Proceedings of Agile Conference, August 24-28, 2009 Chicago, IL, USA. IEEE Computer Society, 245-250.

Debois, P. (2008). Agile Infrastructure and Operations: How Infra-gile are You? Proceedings of Agile Conference, 4-8 Aug. 2008, 202-207.-***A11**

Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A. (2001). Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Trans. on Software Engineering, 27(1):1**-** 12.

Fowler, M. (2009). TechnicalDebtQuadrant. http://www.martinfowler.com/bliki/ TechnicalDebtQuadrant. html [Accessed April 28 2011].

Heidenberg, J. and Porres, I. (2010). Metrics Functions for Kanban Guards. Proceedings of 17th IEEE Int. Conf and Workshops on Engineering of Computer Based Systems, 22-26 March 2010, 306-310.-***A4**

Kitchenham, B. 2004. Procedures for Performing Systematic Reviews. Keele University, Technical Report TR/SE-0401, ISSN:1353-7776..

Klein, J. (2005). How Does the Architect’s Role Change as the Software Ages? 5th Working IEEE/IFIP Conference on Software Architecture, 2005, 141-141. ***A17**

Ktata, O.and Levesque, G. (2010). Designing and Implementing a Measurement Program for Scrum Teams: What do Agile Developers really Need and Want? 3rd C* Conference on Computer Science and Software Engineering, May 19-20, 2010, Montreal, QC, Canada. ACM, 101-107.-***A5**

Lindgren, M., Land, R., Norstrom, C. and Wall, A. (2008). Key Aspects of Software Release Planning in Industry. 19th Australian Conference on Software Engineering, 26-28 March 2008. 320-329.-***A13**

Lindgren, M., Wall, A., Land, R. and Norstrom, C. (2008b). A Method for Balancing Short- and Long-Term Investments: Quality vs. Features. 34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '08, 3-5 Sept. 2008. 175-182.-***A12**

Lutz, M.J. (1993). A Maturing Field on Display at OOPSLA. IEEE Software, 10**,** 113. ***A18**

Mantyla, M.V. and Lassenius, C. 2009. What Types of Defects Are Really Discovered in Code Reviews? *Software Engineering, IEEE Transactions on,* 35**,** 430-448.-***A9**

Neill, C.J. and Laplante, P.A. (2006). Paying Down Design Debt with Strategic Refactoring. Computer, 39**:** 131-134. **-*A16**

O'Connor, D. (2010). Technical Debt in Semiconductor Equipment: It's Time to Pay It Down. Solid State Technology, 53: 34-35.-***A6**

Shull, F. (2011). Perfectionists in a World of Finite Resources. IEEE Software, 28**,** 4-6.-* **A1**

Smith, R. (2009). Computer System Design. Journal of GXP Compliance, 13**,** 44-48. -***A10**

Thibodeau, P. (2011). Counting 'Technical Debt'. Information Age**,** p.64.

Torkar R, Minoves P and Garrigos J (2011). Adopting Free/Libre/Open Source Software Practices, Techniques & Methods for Industrial Use*. J. of the Assoc. for Information Systems, 12**,** 88-122. -***A2**

Tom, E. (2011). A Consolidated Understanding of Technical Debt. Honours Thesis. School of Information Systems, Technology and Management, University of New South Wales, Sydney, 2052, Australia

Wirfs-Brock, R.J. (2008a). Designing Then and Now. IEEE Software, 25**,** 29.-***A14**

Wirfs-Brock, R.J. (2008b). Enabling Change. Software, IEEE, 25**,** 70-71. ***A15**

Wong, W.E., Tse, T.H., Glass, R.L., Basili, V.R. and Chen, T.Y. (2011). Controversy Corner: An Assessment of Systems and Software Engineering Scholars and Institutions (2003-2007 and 2004-2008). J. Syst. Softw., 84**,** 162-168.

Appendix: Extracted Data and Themes

				A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19
ELEMENTS			Code	X	X	X			X	X		X	X						X	X		
			Design / Architecture	X	X	X	X		X	X					X			X		X	X	
			Defects / Issues			X				X										X		
			Documentation	X		X			X													
			Infrastructure											X								
			Testing	X		X			X													
			Unimplemented Features			X							X									
		CONSEQUENCES	Bankruptcy						X													
			Interest	X		X							X		X	X		X		X	X	
			Relationship to Quality		X	X	X						X		X				X			
			Short / Long Term					X					X									
			Difficulty and Risk		X	X	X	X	X	X		X		X			X		X	X		
	CAUSES		Healthy vs. Unhealthy	X				X														X
			Quadrant			X			X													
			Timelines / Scheduling / Velocity	X	X	X	X		X	X	X		X		X	X		X			X	X
			Budget / Resource Constraints						X				X									
			Visibility of Debt				X	X			X		X									

		A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19
STUDY TYPE	Expert Opinion / Secondary Study	X		X	X		X	X	X		X				X	X	X	X	X	
	Primary Study or Experience Report		X			X				X		X	X	X						X
	Primary Study examining Technical Debt																			
DEFINITION / CONCEPTUAL MODEL	Comprehensive / Complete																			
	Incomprehensive / Incomplete	X	X	X	X	X	X	X			X		X	X		X	X	X		X
	Negligible or Absent								X	X		X			X				X	